DIGITAL ARITHMETIC

As the ability to perform computation has increased from the early days of computers to the present so has our knowledge of how to utilize the hardware and software to perform computations. Digital computer arithmetic emerged from that period in two ways: as an aspect of logic design and as the development of efficient algorithms to use the available hardware.

Given that numbers in a digital computer are represented as a string of zeroes and ones and that hardware can perform only a relatively simple and primitive set of Boolean operations, all the arithmetic operations performed are based on a hierarchy of operations that are built upon the very simple ones.

What distinguishes computer arithmetic is its intrinsic relation to technology and the ways things are designed and implemented in a digital computer. This comes from the fact that the value of a particular way to compute, or a particular algorithm, is directly evaluated from the actual speed with which this computation is performed. Therefore, there is a very direct and strong relationship between the technology in which digital logic is implemented to compute and the way the computation is structured. This relationship is one of the guiding principles in the development of computer arithmetic.

For simpler treatment, the subject of computer arithmetic can be divided into number representation, basic arithmetic operations (such as addition, multiplication, and division), and evaluation of functions.

NUMBER REPRESENTATION

The only way to represent information in a digital computer is via a string of bits (i.e., zeroes and ones). The number of bits being used depends on the length of the *computer word,* which is a quantity of bits on which hardware is capable of operating (sometimes also a quantity that is brought to the CPU from memory in a single access). First, we must decide what relationship to use in establishing the correspondence between those bits and a number. Second, we need to make sure that certain properties that exist in the corresponding number system are satisfied and that they directly correspond to the operations being performed in hardware over the taken string of bits.

This relationship is defined by the rule that associates one numerical value designated as X (in the text we will use capital X for the numerical value) with the corresponding bit string designated as x.

$$x = \{x_{n-1}, x_{n-2}, \ldots, x_0\}$$

where

$$x_i \in \{0, 1\}$$

In this case, the associated word (the string of bits) is n bits long.

When for every value X there exists one and only one corresponding bit string x, we define the number system as *nonredundant.* If however, we could have more than one bit string x that represents the same value X, the number system is *redundant.*

Most commonly we use numbers represented in a *weighted* number system where a numerical value is associated with the bit string x according to the equation:

$$x = \sum_{i=0}^{n-1} x_i \times w_i$$

where

$$w_0 = 1$$

and

$$w_i = w_{i-1} \times r_{i-1}$$

The value r_i is an integer designated as *radix,* and in a nonredundant number system it is an integer equal to the number of allowed values for x_i. In general, x_i could consist of more than one bit. The numerical value associated with x is designated as the *explicit value* of x. In conventional number systems, the radix r_i is the same positive integer for all the digit positions x_i and with the canonical set of digit values:

$$\Sigma_i = \{0, 1, 2, 3, \ldots, r_i - 1\} \quad \text{for} \quad (0 \le i \le n-1)$$

An example of a weighted number system with a mixed-radix would be the representation of time in weeks, days, hours, minutes, and seconds with a range for representing 100 weeks:

$$r = 10, 10, 7, 24, 60, 60$$

In digital computers, the radixes encountered are 2, 4, 10, and 16, with 2 being the most commonly used radix.

The digit set x_i can be redundant and nonredundant. If the number of different values x_i can assume is $n_x \le r$, then we have a nonredundant digit set. Otherwise, if $n_x > r$, we have a redundant digit set. Use of the redundant digit set has its advantages in efficient implementation of algorithms (multiplication and division in particular).

Other number representations of interest are *nonweighted* number systems where the relative position of the digit does not affect the weight, so that the appropriate interchange of any two digits will not change the value x. The best example of such number system is the Residue Number System (RNS).

We also define the *explicit value* x_e and *implicit value* X_i of a number represented by a bit string x. The implicit value is the only value of interest to the user, whereas the explicit value provides the most direct interpretation of the bit string x. Mapping of the explicit value to the implicit value is obtained by an arithmetic function that defines the number representation used. It is a task of the arithmetic designer to devise algorithms that effect the correct implicit value of the result for the operations on the operand digits representing the explicit values. In other words, the arithmetic algorithm must satisfy the *closure* property. The relationship between the implicit value and the explicit value is best illustrated by Table 1 (1).

Table 1. The Relationship Between the Implicit Value and the Explicit Value for $X = 11011$, $r = 2$

Implied Attributes: Radix Point, Negative Number Representation	Expression for Implicit Value X_i as a Function of Explicit Value x_e	Numerical Implicit Value X_i (in decimal)
Integer, magnitude	$X_i = x_e$	27
Integer, two's complement	$X_i = -2^5 + x_e$	−5
Integer, one's complement	$X_i = -(2^5 - 1) + x_e$	−4
Fraction, magnitude	$X_i = 2^{-5} x_e$	27/32
Fraction, two's complement	$X_i = 2^{-4}(-2^{-5} + x_e)$	−5/16
Fraction, one's complement	$X_i = 2^{-4}(-2^{-5} + 1 + x_e)$	−4/16

Source: Adapted from Ref. 1.

Representation of Signed Integers

The two most common representations of signed integers are Sign and Magnitude (SM) representation and True and Complement (TC) representation. Even though SM representation might be easier to understand and convert to and from, it has its own problems. Therefore, TC representation is more commonly used.

Sign and Magnitude Representation. In SM representation, signed integer X_i is represented by sign bit x_s and magnitude x_m (x_s, x_m). Usually 0 represents a positive sign (+), and 1 represents a negative sign (−). The magnitude of the number x_m can be represented in any way chosen for the representation of positive integers. The disadvantage of SM representation is that two representations of zero exist, positive and negative zero: $x_s = 0$, $x_m = 0$ and $x_s = 1$, $x_m = 0$.

True and Complement Representation. In TC representation, there is no separate bit used to represent the sign. Mapping between the explicit and implicit value is defined as

$$X_i = \begin{cases} x_e & x_e < \dfrac{C}{2} \\ x_e - C & x_e > \dfrac{C}{2} \end{cases}$$

The illustration of the TC mapping is given in Table 2 (2). In this representation, positive integers are represented in the

Table 2. True and Complement Mapping

x_e	X_i
0	0
1	1
2	2
—	—
—	—
$C/2 - 1$	$C/2 - 1$
$C/2 + 1$	$-(C/2 - 1)$
—	—
—	—
$C - 2$	−2
$C - 1$	−1
C	0

Table 3. Mapping of the Explicit Value x_e into RC and DRC Number Representations

x_e	X_i (RC)	X_i (DRC)
0	0	0
1	1	1
2	2	2
—	—	—
—	—	—
$\frac{1}{2}r^n - 1$	$\frac{1}{2}r^n - 1$	$\frac{1}{2}r^n - 1$
$\frac{1}{2}r^n$	$-\frac{1}{2}r^n$	$-(\frac{1}{2}r^n - 1)$
—	—	—
—	—	—
—	—	—
$r^n - 2$	-2	-1
$r^n - 1$	-1	0

True Form, whereas negative integers are represented in the *Complement Form.*

With respect to how the complementation constant C is chosen, we can further distinguish two representations within the TC system. If the complementation constant is chosen to be equal to the range of possible values taken by x_e, $C = r^n$ in a conventional number system where $0 \le x_e \le r^n - 1$, then we have defined the *Range Complement* (RC) system. If, on the other hand, the complementation constant is chosen to be $C = r^n - 1$, we have defined the *Diminished Radix Complement* (DRC), [also known as *Digit Complement* (DC)] number system. Representations of the RC and DRC number representation systems are shown in Table 3.

As can be seen from Table 3, the RC system provides for one unique representation of zero because the complementation constant $C = r^n$ falls outside the range. There are two representations of zero in the DRC system, $x_e = 0$ and $r^n - 1$. The RC representation is not symmetrical, and it is not a closed system under the change of sign operation. The range for RC is $[-\frac{1}{2}r^n, \frac{1}{2}r^n - 1]$. The DRC is symmetrical and has the range of $[-(\frac{1}{2}r^n - 1), \frac{1}{2}r^n - 1]$.

For the radix $r = 2$, RC and DRC number representations are commonly known as *two's complement* and *one's complement* number representation systems. Those two representations are illustrated by an example in Table 4 for the range of values $-(4 \le X_i \le 3)$.

Table 4. Two's Complement and One's Complement Representation

	Two's Complement	$C = 8$	One's Complement	$C = 7$
X_i	x_e	X_i Two's Complement	x_e	X_i One's Complement
3	3	011	3	011
2	2	010	2	010
1	1	001	1	001
0	0	000	0	000
−0	0	000	7	111
−1	7	111	6	110
−2	6	110	5	101
−3	5	101	4	100
−4	4	100	3	—

ALGORITHMS FOR ELEMENTARY ARITHMETIC OPERATIONS

The algorithms for the arithmetic operation are dependent on the number representation system used. Therefore, their implementation should be examined for each number representation system separately, given that the complexity of the algorithm, as well as its hardware implementation is dependent on it.

Addition and Subtraction in Sign and Magnitude System

In the SM number system, addition/subtraction is performed on pairs (u_s, u_m) and (w_s, w_m) resulting in a sum (s_s, s_m), where u_s and w_s are sign bits and u_m and w_m are magnitudes. The algorithm is relatively complex because it requires comparisons of the signs and magnitudes. Extending the addition algorithm in order to perform subtraction is relatively easy because it involves only a change of the sign of the operand being subtracted. Therefore, we will consider only the addition algorithm.

The algorithm can be described as

if $u_s = w_s$ (signs are equal) **then**

$$s_s = u_s \quad \text{and} \quad s_m = u_m + w_m$$

(the operation includes checking for the overflow)

if $u_s \neq w_s$ **then**

if $u_m > w_m : s_m = u_m - w_m, s_s = u_s$

else $s_m = w_m - u_m, s_s = w_s$

Addition and Subtraction in True and Complement System

Addition in the TC system is relatively simple. It is sufficient to perform modulo addition of the explicit values, therefore,

$$s_e = (u_e + w_e) \bmod C$$

Proof is omitted.

In the RC number system, this is equivalent to passing the operands through an adder and discarding the carry-out of the most significant position of the adder, which is equivalent to performing the modulo addition (given that $C = r^n$).

In the DRC (DC) number system, the complementation constant is $C = r^n - 1$. Modulo addition in this case is performed by subtracting r^n and adding 1. It turns out that this operation can be performed by simply passing the operands through an adder and feeding carry-out from the most significant digit position into the carry-in at the least significant digit position. This is also called addition with end-around-carry.

To subtract two numbers, simply change the sign of the operand to be subtracted and then proceed with the addition operation.

Change of Sign Operation

The change of sign operation involves the following operation:

$$W_i = -Z_i$$

$$w_e = (-z_e) = (-z_e) \bmod C = C - Z_i \bmod C = C - z_e$$

which means that change of sign operation consists of subtracting the operand z_e from the complementation constant C. In the DRC (DC) system, complementation is performed simply by complementing each digit of the operand Z_i with respect to $r - 1$. In case of $r = 2$, this result is the simple inversion of bits.

In case of RC system, the complementation is performed by complementing each digit with respect to $r - 1$ and adding one to the result.

Implementation of Addition

Carry Look-Ahead Adder. The first significant speed improvement in the implementation of a parallel adder was a Carry Look-Ahead Adder (CLA) developed by Weinberger and Smith in 1963 (4). The CLA is one of the fastest schemes used for adding two numbers even today, given that the delay incurred to add two numbers is logarithmically dependent on the size of the operands (delay = log N). The concept of CLA is illustrated in Fig. 1. For each bit position of the adder, a pair of signals (p_i, g_i) is generated in parallel. It is possible to generate local carries using (p_i, g_i) as seen in the equations. Those signals are designated as *pi* (carry-propagate) and *gi* (carry-generate) because they take part in the propagation and generation of carry signal C_{i-1}. However, each bit position requires an incoming signal C_{i-1} in order to generate the outgoing carry C_i. This makes the addition slow because the carry signal must ripple from stage to stage as shown in Fig. 1(a). The adder can be divided into groups and the carry-generate and carry-propagate signals can be calculated for the entire group (G, P). This will take an additional time equivalent to AND-OR delay of the logic. However, now we can calculate each group's carry signals in an additional AND-OR delay. For the generation of the carry signal from the adder, only the incoming carry signal into the group is now required.

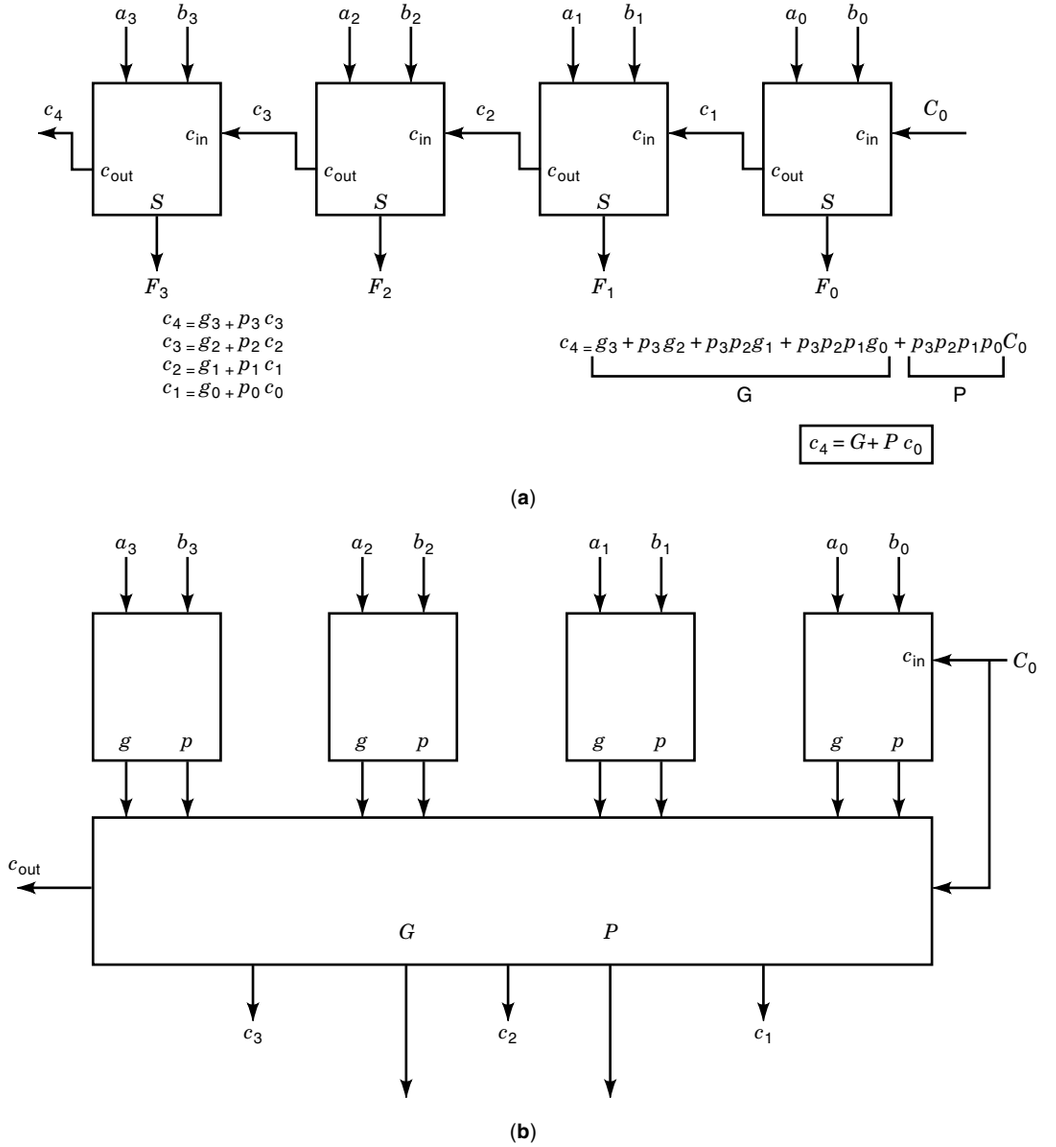

Figure 1. The Carry Look-Ahead Adder structure: (a) generation of carry, generate, and propagate signals; and (b) generation of Group signals, G, P and intermediate carries.

Therefore, the rippling of the carry is limited only to the groups. In the next step, we may calculate, generate, and propagate signals for the group of groups (G^*, P^*) and continue in that fashion until we have only one group left generating the C_{out} signal from the adder. This process will terminate in log N steps, given that we generate a tree structure for a generation of carries. The computation of carries within the groups is done individually as illustrated in Fig. 1(a), and this process requires only the incoming carry into the group (3).

The logarithmic dependence on the delay (delay = log N) is valid only under the assumption that the gate delay is constant without depending on the fan-out and fan-in of the gate. In practice, this is not true. Even when the bipolar technology (which does not exhibit strong dependence on the fan-out) is used to implement CLA structure, the further expansion of the carry-block is not possible given the practical limitations of the fan-in of the gate.

In CMOS technology, this situation is much different, given that CMOS gate has strong dependency not only on fan-in but on fan-out as well. This limitation takes away many of the advantages gained by using the CLA scheme (6). However, by clever optimization of the critical path and appropriate use of dynamic logic, the CLA scheme can still be advantageous, especially for the adders of a larger size.

Conditional-Sum Addition. Another one of the fast schemes for adding two numbers, which predates CLA, is Conditional-Sum Addition (CSA) proposed by Sklansky in 1960 (5). The essence of the CSA scheme is the realization that we can add two numbers without waiting for the carry signal to be available. Simply, the numbers are added in two instances: one assuming $C_{in} = 0$ and the other assuming $C_{in} = 1$. The results: Sum_0, Sum_1 and $Carry_0$, $Carry_1$ are presented at the input of a multiplexer. The final values are being selected when C_{in} arrives at the "select" input of a multiplexer. As in CLA, the input bits are divided into groups that are added "conditionally."

It is apparent that starting from the Least Significant Bit (LSB) position, the hardware complexity starts to grow rapidly. Therefore, in practice, the full-blown implementation of the CSA is not often seen.

However, the idea of adding the Most Significant Bit (MSB) portion conditionally and selecting the results after the carry-in signal is computed in the LSB portion is attractive. Such a scheme (which is a subset of CSA) is known as Carry-Select Adder. A 26-bit Carry-Select Adder consisting of two 13-bit portions is shown in Fig. 2.

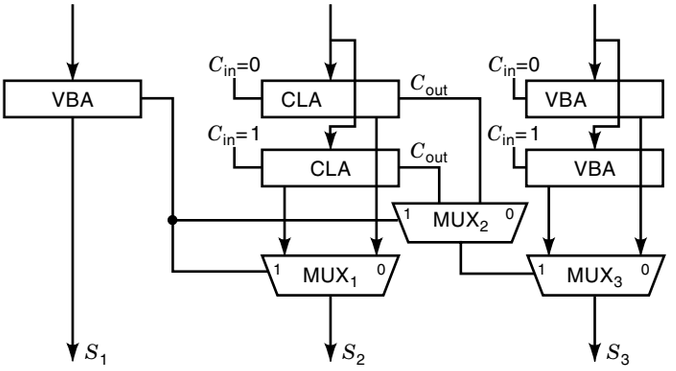

Figure 2. 26-bit Carry-Select Adder.

Multiplication Algorithm

The multiplication operation is performed in a variety of forms, in hardware and software. In the beginning of the computer development, any complex operation was usually programmed in software or coded in the microcode of the machine. Some limited hardware assistance was provided. Today you are more likely to find full hardware implementation of the multiplication based on speed and the reduced cost of hardware. However, in all of them, multiplication shares the basic algorithm with some adaptations and modifications to particular implementation and number system used. For simplicity, we will describe a basic multiplication algorithm that operates on positive n-bit long integers X and Y resulting in the product P, which is $2n$ bit long:

$$P = XY = X \times \sum_{i=0}^{n-1} y_i r^i = \sum_{i=0}^{n-1} X \times y_i r^i$$

This expression indicates that the multiplication process is performed by summing n terms of a *partial product:* $X \times y_i r^i$. This product indicates that the ith term is obtained by a simple arithmetic left shift of X for the i positions and multiplication by the single digit y_i. For the binary radix $r = 2$, y_i is 0 or 1 and multiplication by the digit y_i is very simple to perform. The addition of n terms can be performed at once, by passing the partial products through a network of adders (which is the case of full hardware multiplier) or sequentially, by passing the partial product through an adder n times. The algorithm to perform multiplication of X and Y can be described as

$$p^{(0)} = 0$$

$$p^{(j+1)} = \frac{1}{r}(p^{(j)} + r^n X y_j) \text{ for } j = 0, \ldots, n-1$$

It can be easily proved that this recurrence results in $p^{(n)} = XY$.

Various modifications of the multiplication algorithm exist, one of the most famous is the *Modified Booth Recoding Algorithm* described by Booth in 1951. This algorithm allows for the reduction of the number of partial products, thus speeding up the multiplication process. Generally speaking, the Booth algorithm is a case of using the redundant number system with the radix higher than 2.

Implementation of Multiplication Algorithm

The speed with which the multiplication operation is performed is of utmost importance in Digital Signal Processors (DSP) today as well as in the general-purpose processors. Therefore, research in building a fast parallel multiplier has been going on since the first such paper was published by Wallace in 1964 (7). In his historic paper, C. S. Wallace introduced a way of summing the partial product bits in parallel using a tree of Carry-Save Adders, which became generally known as a *Wallace Tree.*

A suggestion for speed improvement of such process of adding partial product bits in parallel followed in the paper pub-

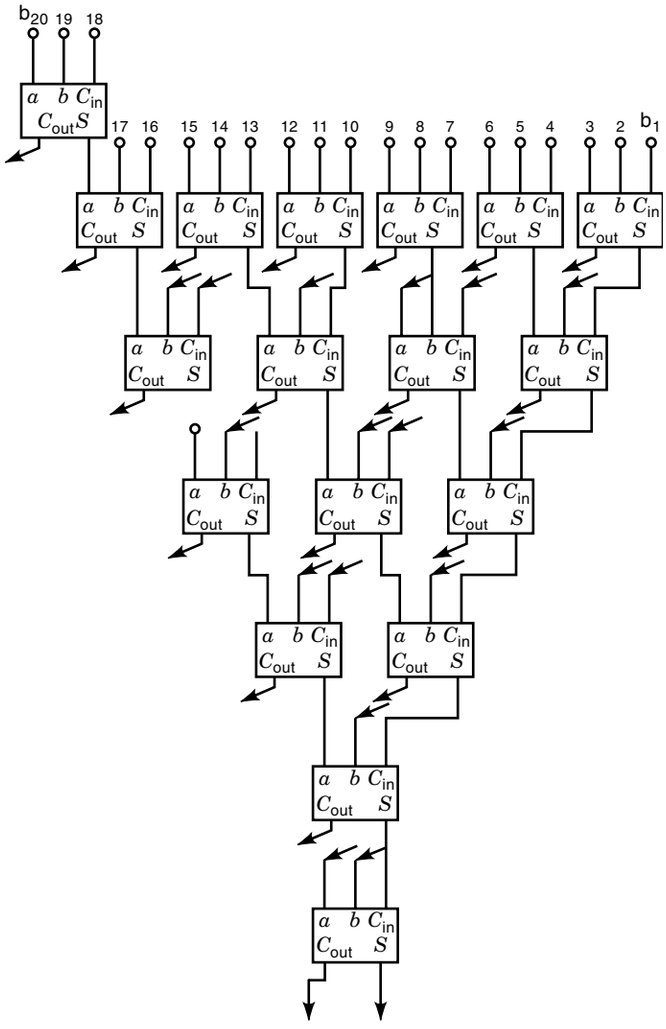

Figure 3. Wallace Tree.

lished by Dadda (8). In this paper, Dadda introduced the notion of a counter structure that will take a number of bits p in the same bit position (of the same "weight") and output a number q that represents the count of ones in the input. Dadda has introduced a number of ways to compress the partial product bits using such a counter, which later became known as *Dadda's counter*.

The quest for making the parallel multiplier even faster continued for almost 30 years. The search for producing a fastest "counter" did not result in a general structure that yielded a faster partial product summation than that which used Full-Adder (FA) cell or 3 : 2 counter. Therefore, using a Wallace Tree was almost prevalent in the implementation of the parallel multipliers. In 1981 Weinberger disclosed a structure that he called the 4-2 carry-save module. This structure contained a combination of FA cells in an intricate interconnection structure, which yielded faster partial product compression than the use of 3 : 2 counters.

The structure actually compresses five partial product bits into three; however, it is connected in such a way that four of the inputs are coming from the same bit position of the weight j whereas one bit is fed from the neighboring position $j - 1$ (known as carry-in). The output of such a 4-2 module consists of one bit in the position j and two bits in the position $j + 1$. This structure does not represent a counter (even though it became erroneously known as a 4-2 counter), but rather a compressor, which would compress four partial product bits into two (while using one bit laterally connected between adjacent 4-2 compressors). The efficiency of such a structure is higher (it reduces the number of partial product bits by one-half). The speed of such a 4-2 compressor has been determined by the speed of three XOR gates in series (in the redesigned version of 4-2 compressor) making such a scheme more efficient than the one using 3 : 2 counters in a regular Wallace Tree. The other equally important feature of using 4-2 compressor is that the interconnections between such cells follow a more regular pattern than in the case of a Wallace Tree.

Booth Encoding. Various ways for reducing the number of partial products exist; one of the most famous is the Booth Recoding Algorithm described by Booth in 1951 (9). This algorithm allows for the reduction of the number of partial products by roughly one-half, thus speeding up the multiplication process. Generally speaking, the Booth algorithm is a case of using the redundant number system with the radix higher than 2.

Booth's algorithm (9) is widely used in the implementation of hardware or software multipliers because its application makes it possible to reduce the number of partial products. It can be used for both sign-magnitude numbers as well as two's complement numbers with no need for a correction term or a correction step.

A modification of the Booth algorithm was proposed by Mac Sorley (10) in which a triplet of bits instead of two bits is scanned. This technique has the advantage of reducing the number of partial products by half regardless of the inputs. This result is summarized in Table 5.

The recoding is performed within two steps: encoding and selection. The purpose of the encoding is to scan the triplet of bits of the multiplier and define the operation to be performed on the multiplicand, as shown in Table 1. This method is actually an application of a sign-digit representation in radix 4. The Booth-MacSorley algorithm, usually called the Modified

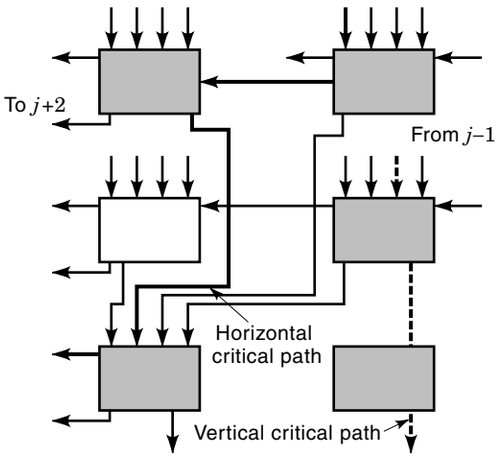

Figure 4. 4 : 2 Compressor.

Table 5. Modified Booth Recoding

$x_{i+2}x_{i+1}x_i$	Add to Partial Product
000	$+0Y$
001	$+1Y$
010	$+1Y$
011	$+2Y$
100	$-2Y$
101	$-1Y$
110	$-1Y$
111	$-0Y$

Booth algorithm or simply the Booth algorithm, can be generalized to any radix.

Booth recoding necessitates the internal use of two's complement representation in order to efficiently perform subtraction of the partial products as well as additions. However, the floating-point standard specifies sign magnitude representation, which is followed by most of the nonstandard floating-point numbers in use today. The advantage of Booth recoding is that it generates only half of the partial products as compared to the multiplier implementation, which does not use Booth recoding. However, the benefit achieved comes at the expense of increased hardware complexity. Indeed, this implementation requires hardware for the encoding and for the selection of the partial products ($0, \pm Y, \pm 2Y$). An optimized encoding is shown in Fig. 5.

Division Algorithm

Division is a more complex process to implement because, unlike multiplication, it involves *guessing* the digits of the quotient. Here, we will consider an algorithm for division of two positive integers designated as dividend Y and divisor X, which result in a quotient Q and an integer remainder Z according to the relation given by

$$Y = XQ + Z$$

In this case, the dividend contains $2n$ integers, and the divisor has n digits in order to produce a quotient with n digits.

The algorithm for division is given with the following recurrence relationship (2):

$$z^{(0)} = Y$$
$$z^{(j+1)} = rz^{(j)} - Xr^n Q_{n-1-j} \text{ for } j = 0, \ldots, n-1$$

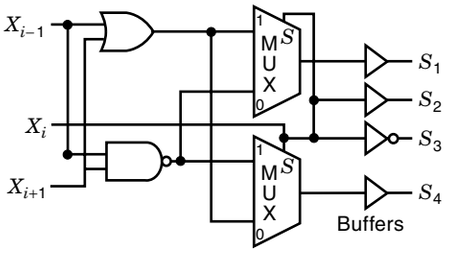

Figure 5. Booth Encoder.

This recurrence relation yields

$$z^{(n)} = r^n(Y - XQ)$$
$$Y = XQ + z^{(n)}r^{-n}$$

which defines the division process with remainder $Z = z^{(n)}r^{-n}$. The selection of the quotient digit is done by satisfying that $0 \le Z < X$ at each step in the division process. This selection is a crucial part of the algorithm and the best known are *restoring* and *nonrestoring* division algorithms. In the former algorithm, the value of the *tentative partial remainder* $z^{(j)}$ is restored after the wrong guess is made of the quotient digit q_j. In the latter, this correction is not done in a separate step but rather in the step following. The best-known division algorithm is the so-called SRT algorithm, which was independently developed by Sweeney, Robertson, and Tocher. Algorithms for higher radix were further developed by Robertson and his students, most notably Ercegovac.

FURTHER READING

For more information about specific arithmetic algorithms and their implementation, consult: Kai Hwang, *Computer Arithmetic: Principles, Architecture and Design,* New York: John Wiley & Sons, 1979. Also see, E. Swartzlander, *Computer Arithmetic,* Vols. I & II, Los Alamitos, CA: IEEE Computer Society Press, 1980. Publications in *IEEE Transactions on Electronic Computers* and *Proceedings of the Computer Arithmetic Symposiums* by various authors are also very good sources for detailed information on particular algorithm or implementation.

DEFINING TERMS

Algorithm. The decomposition of the computation into subcomputations with an associated precedence relation that determine the order in which these subcomputations are performed (2).

Number Representation System. A defined rule that associates one numerical value x_e with every valid bit string x.

Nonredundant Number System. The system where for each bit string there is one and only one corresponding numerical value x_e.

Redundant Number System. The system in which the numeric value x_e could be represented by more than one bit string.

Explicit Value x_e. A value associated with the bit string according to the rule defined by the number representation system being used.

Implicit Value X_i. The value obtained by applying the arithmetic function defined for the interpretation of the explicit value x_e.

Carry Look-Ahead Adder. An implementation technique of addition that accelerate the propagation of the carry signal, thus increasing the speed of addition operation.

Wallace Tree. A technique for summing the partial product bits of a parallel multiplier in a carry-save fashion using full-adder cells.

Dadda's Counter. A generalized structure used to produce a number (count) representing the number of bits that are "one." It is used for efficient reduction of partial product bits.

4 : 2 Compressor. A structure used in the partial product reduction tree of a parallel multiplier for achieving faster and more efficient reduction of the partial product bits.

Booth-MacSorley Algorithm. Algorithm used for recoding of the multiplier such that the number of partial products is roughly reduced by a factor of 2. It is a special case of the application of the redundant number system to represent the multiplier.

SRT Algorithm. Algorithm for division of binary numbers, which uses redundant number representation.

BIBLIOGRAPHY

1. A. Avizienis, Digital computer arithmetic: A unified algorithmic specification, *Symp. Comput. Automata,* Polytechnic Institute of Brooklyn, April 13–15, 1971.
2. M. Ercegovac, Arithmetic algorithms and processors, *Digital Systems and Hardware/Firmware Algorithms,* New York: Wiley, 1985.
3. S. Waser and M. Flynn, *Introduction to Arithmetic for Digital Systems Designers,* New York: Holt, Rinehart and Winston, 1982.
4. Weinberger and J. L. Smith, *A Logic for High-Speed Addition,* Circulation 591, National Bureau of Standards, pp. 3–12, 1958.
5. Sklanski, Conditional-sum addition logic, *IRE Trans. Electron. Comput.,* **EC-9**: 226–231, 1960.
6. V. G. Oklobdzija and E. R. Barnes, Some optimal schemes for ALU implementation in VLSI technology, *Proc. 7th Symp. Comput. Arithmetic,* University of Illinois, Urbana, IL, June 4–6, 1985.
7. C. S. Wallace, A suggestion for a fast multiplier, *IEE Trans. Electron. Comput.,* **EC-13**: 14–17, 1964.
8. L. Dadda, Some schemes for parallel multipliers, *Alta Frequenza,* **34**: 349–356, 1965.
9. A. D. Booth, A signed binary multiplication technique, *Quart. J. Mech. Appl. Math.,* **IV**, 1951.
10. O. L. Mac Sorley, High speed arithmetic in binary computers, *Proc. IRE,* **49** (1): January 1961.

VOJIN G. OKLOBDZIJA
University of California

DIGITAL ARITHMETIC. See BOOLEAN FUNCTIONS.